Processing Regular Path Queries on Arbitrarily Distributed Data

Processing Regular Path Queries on Arbitrarily Distributed Data

Alan Davoust and Babak Esfandiari

Dept of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada

Abstract. Regular Path Queries (RPQs) are a type of graph query where answers are pairs of nodes connected by a sequence of edges matching a regular expression.
We study the techniques to process such queries on a distributed graph of data.
While many techniques assume the location of each data element (node or edge) is known, when the components of the distributed system are autonomous, the data will be arbitrarily distributed, or *non-localized*.
We compare query processing strategies for this setting analytically and empirically, using biomedical data and meaningful queries. We isolate query-dependent cost factors and present a method to choose between strategies, using new query cost estimation techniques.

1 Introduction

Regular Path Queries (RPQs) were first introduced as part of a query language for graph databases [7, 6], and gained particular interest as a way of querying the distributed graph formed by the World Wide Web pages and hyperlinks [1, 2, 17]. More recently, there has been renewed interest in RPQs with the development of the Semantic Web, after their introduction in version 1.1 of the SPARQL query language ("property paths").

In this paper, we study the problem of processing regular path queries over distributed data. More specifically, we consider the situation where the data is distributed in an arbitrary and uncontrolled manner across many network locations. These locations may be peers in a peer-to-peer (P2P) network (e.g. systems following the model of Piazza [11] or Edutella [20]), or servers on the Web of Data: the situations are very similar.

In other words, given a node in the graph represented by data at location L_1, data representing adjacent nodes (or incident edges) may be found at L_1 but also possibly at arbitrary other locations $L_i, L_j, \ldots, L_k$. We refer to such a distribution model as *non-localized data*.

While this assumption (or lack thereof) may seem extreme, it simply reflects the fact that this data is published by *autonomous* entities, that can freely choose which data they host.

This is the case for example for the graph of Linked Data on the Web (LDOW). Consider the URI `http://dbpedia.org/resource/Edinburgh` (we will denote the prefix as `dbpedia:` for short): incident edges to this node are RDF triples describing this resource, in the form (`dbpedia:Edinburgh`, $?p$, $?o$), or ($?s$, $?p$,`dbpedia:Edinburgh`). Such triples will of course be published primarily by the site `dbpedia.org`; however, a fundamental best practice of LDOW is to link resources from one dataset to another, which means that it is not only possible but *encouraged* to publish such triples in other locations. Accordingly, triples of the form ($?s$, `owl:sameas`, `dbpedia:Edinburgh`) can be found at 23 or more other locations[1] on the Web of Data, including `fu-berlin.de`, `bbc.co.uk`, `musicbrainz.org`, `wikidata.org`, and many others.

To further complicate matters, triples – or entire graphs of data – may be replicated at different sites: for example, the *Drugbank* dataset is entirely contained in the *DERI Health Care and Life Science* dataset[2] [22].

The processing of RPQ has not previously been studied in the setting of non-localized data. A naïve solution would be to simply download all the available data and process the query locally. However, with large datasets (e.g. the fast-growing LDOW cloud), this is impractical. Several distributed query processing algorithms were proposed for the graph of Web pages and hyperlinks [2, 8] and for distributed graph databases [23], but these algorithms rely on the data being localized. In addition, they cannot be used for Regular Path Queries with Inverse [4, 3], a useful extension of RPQ where edges can be traversed in both directions.

This leaves centralized processing techniques, where the data sources are dynamically accessed for each query. Ideally, we would like to collect exactly the data needed to answer the query. However, for RPQ it is quite difficult to know in advance which data will be needed, due to language features such as wildcards and transitive closures, and the lack of fixed schemas in graph data.

The data may be collected either in a single step before executing the query locally, or else on the fly during query execution [13, 12]. Ladwig and Tran [16] refer to the former approach as "top-down", and to the latter as "bottom-up".

We show that neither strategy consistently outperforms the other, and in the worst case, both strategies may result in retrieving the entire graph of data, which is problematic.

However, in practice we can expect the majority of queries to be more *selective*, and thus tractable. Therefore, ideally we would like to be able to (at least probabilistically) identify those tractable queries, and secondly have a method to determine the best processing strategy for a given query.

We show that the optimal strategy choice depends on the selectivity of the query, and propose two techniques to estimate the "selectivity" of queries based on a sample of the data. We evaluate our techniques against a real dataset from the biomedical domain and a set of meaningful queries.

[1] the site `sameas.org` lists over 300 alternative URIs identifying the scottish city, hosted by 23 different top-level domains, i.e. independent locations

[2] http://hcls.deri.org:8080/openrdf-sesame/repositories/hclskb

The rest of this paper is organized as follows. In section 2 we provide some background definitions and algorithms for RPQ in general. In section 3 we evaluate existing query processing techniques for the specific setting of non-localized data. In section 4, we evaluate and compare two query execution strategies against the biomedical data, and determine conditions to choose between the strategies. Finally, we present our query cost evaluation techniques in section 5, and we draw some conclusions in section 6.

2 Definitions and Algorithms for RPQs

Informally, the idea of a regular path query is to find pairs of nodes in a graph of data, such that the path (sequence of edges) from one node to the other matches a given regular expression. Two variations of RPQs have been considered in the literature: multi-source queries, that return every pair of nodes matching the query, and single-source queries, where a single "start node" is given.

2.1 Notations and Definitions

All queries are applied to an edge-labeled directed graph $G_D = \langle V, E \rangle$, where $V = \{v_0, v_1, \ldots, v_N\}$ are nodes and $E \subset V \times \Delta \times V$ are edges labeled from a set of labels $\Delta = \{\delta_i\}$.

A path in G_D from a node v_0 to a node v_k is a sequence of adjacent edges $(v_0, \delta_1, v_1), (v_1, \delta_2, v_2), \ldots, (v_{k-1}, \delta_k, v_k)$ starting at node v_0 and ending at node v_k. Note that this definition permits repeated edges or nodes, which is the standard semantics used in the context of RPQ.

The notation $v_0 \xrightarrow{w} v_k$ indicates that there exists a path from v_0 to v_k such that the sequence of edge labels $\delta_1, \delta_2, \ldots, \delta_k$ along this path forms a word w. If r is a regular expression, we note $L(r)$ the regular language defined by r.

Definition 1 (Multi-source Query). *A multi-source query Q_r is defined by a regular expression r over Δ. When Q_r is applied to the graph G_D, the answers to Q_r are defined as follows:*

$$Ans(Q_r, G_D) = \{(v_i, v_j) \in V \times V | v_i \xrightarrow{w} v_j, w \in L(r)\}$$

Definition 2 (Single-source Query). *A single-source query Q_{r,v_0}, applicable to the graph G_D, is defined by a regular expression r over Δ, and a distinguished node v_0 of G_D. The answers to Q_r are defined as follows:*

$$Ans(Q_{r,v_0}, G_D) = \{v_j \in V | v_0 \xrightarrow{w} v_j, w \in L(r)\}$$

2.2 Basic RPQ Algorithm

The main algorithm to answer such queries was sketched in a 1989 paper by Mendelzon and Wood [18].

We detail their general algorithm, which is the basis of most other approaches. We will refer to this algorithm as the *product automaton algorithm*, or PAA:

1. build a finite automaton A_1 associated with the regular expression r. The initial state of A_1 is q_0, the accepting states are $\{qf_i\}$.
2. consider the graph of data as an automaton A_2 (considering nodes as states, and edges as transitions), and compute the cross-product of the automata $A_p = A_1 \times A_2$.
3. – **(single-source RPQ)**: the initial node in the graph is set (N_0): search A_p from the initial state (q_0, N_0) to find all reachable accepting states (qf_k, N_j). All nodes (N_j) are answers to the single-source query.
   – **(multi-source RPQ)**: we are looking for all pairs of nodes related by the regular path: search A_p from all initial states (q_0, N_i) to find all reachable accepting states (qf_k, N_j). All pairs of nodes (N_i, N_j) are answers to the multi-source query.

For step 3, any graph search algorithm can be used, such as breadth-first or depth-first.

Optimizations Several optimizations to the PAA algorithm have been studied in the literature. The first one [8] helps prune the search of the product automaton A_p, while in the second case [14, 15, 27] the main query is split into smaller subqueries that are also executed using the PAA. We can therefore consider the PAA algorithm to be the fundamental basis of RPQ processing, and in the rest of this paper we will focus on adapting this algorithm to the distributed setting, without further consideration for these optimizations.

2.3 Complexity

For database query languages, the traditional parameters considered in a complexity study are the *data size* and the *query size* [26].

The size of the data graph has two parameters, the number of nodes $|V|$ and the number of edges $|E|$. For a regular path query, the query size is the number of characters and operators in the regular expression [19], which we will note m.

Based on these parameters, the cost of the PAA algorithm is the cost of building the query automaton, plus the cost of building and searching the product automaton. In practice, only a subset of the product automaton will be searched and built on the fly. However, in the worst case the full product automaton is reachable and must be built and searched.

A non-deterministic query automaton can be built in $O(m)$ time and has $O(m)$ states [19]. The product automaton will have $O(|V|.m)$ states and $O(|E|.m)$ transitions.

The complexity of a graph search (BFS or DFS) over this automaton, and thus the total complexity of the PAA is therefore $O((|E| + |V|).m)$.

3 RPQ Processing on Distributed Data

In this section, we present existing techniques to process RPQ on distributed data, and we discuss their applicability to the case of RPQ on non-localized data.

3.1 Distributed Query Execution

The main distributed query processing strategy is based on the idea of "query shipping". In this strategy, first detailed in [2] and applied to the single-source RPQ on the graph of Web documents, during the graph traversal the query is shipped from one Web server to another, as hyperlinks (edges) are traversed in the PAA algorithm. This works for the Web graph because the nodes are Web pages containing all the outgoing edges from this node, and because hyperlinks identify their targetby its network location.

A similar idea is used in [23] to process multi-source RPQ in a P2P graph database system, where data has the same localization properties as the Web graph.

3.2 Query decomposition

As the query shipping strategy may cause a large amount of traffic in shipping queries back and forth between sites, a better solution may be the query decomposition technique proposed by Suciu [24], which allows a query to be answered by exchanging only one pair of messages (subquery-answer) between the querying site and each other site.

The idea of this technique is to anticipate the query shipping by enumerating all the possible subqueries that might need to be shipped between sites, and send them straightaway to all the sites. Each site then processes each subquery from all "incoming nodes", i.e. nodes that have incoming edges from other sites. The results of all these subqueries can then be collected at the querying site to reconstruct the sub-graph traversed in the query execution.

However, this strategy requires distinguishing "local" and "outgoing" edges in the graph of data, which is possible only with a localization model similar to the Web graph.

3.3 SPARQL Query Processing Approaches

In the LDOW context, there have been a number of proposals for *client-side* SPARQL query processing[13, 12, 25], where clients dynamically answer queries using data from multiple remote sources.

The available techniques largely ignore RPQ, which were only recently incorporated into SPARQL in version 1.1. Without RPQ, query planning techniques can be adapted from the domain of relational databases, and the key issue is the ordering of JOIN operations, and of their data retrieval steps. However, these query planning techniques are primarily designed for queries with a finite number of JOINs. For RPQ, a path defined with a Kleene closure (*) may involve an arbitrary number of edge traversals, which are essentially JOINs, if the data is represented by a table listing the edges of the graph (e.g. RDF triples).

For single-source queries, the obvious solution is then to start from the given start node, and compute the JOINs (i.e. traverse edges from this node) iteratively, potentially retrieving new data with each iteration.

Intuitively, this technique retrieves only the data needed for the query, but it may require many separate requests.

This technique has been described as "bottom-up" [16], and can be opposed to a "top-down" approach, which consists instead in retrieving all the data relevant to the query in a single first step, before executing the query locally. The advantage of the "top-down" approach is that it requires a single query to each source, but the disadvantage is that in that initial step, it is much more difficult to pinpoint which data will actually be needed for the query.

In the next section, we will analyze a simple selection technique, and compare it with the "bottom-up" approach.

3.4 Query Processing for Non-localized Data

The query shipping technique described in section 3.1 is clearly inapplicable to the context of non-localized data. An additional disadvantage of this technique is that it requires every participating site (every data source) to support the required algorithm.

The query decomposition technique (section 3.2) requires each site to know which nodes and edges in the local data graph are "incoming" or "outgoing", which is non-trivial for non-localized data. The nodes could periodically exchange this information somehow, but this would require additional specialized protocols. We defer the design and analysis of such protocols to future work.

However, the goal of sending a single subquery to each site can be met by a simple "top-down" approach which we will call S1, where data is selected on the basis of the edge labels appearing in the query. For example, if the query is defined by the regular expression a^*bb, then only the edges labeled a or b will possibly be traversed during the query processing.

Once the possibly relevant data has been retrieved, the graph of data can be reassembled by the querying agent and processed locally. The main problem with this technique is that if the query contains any wildcard edge labels, then the entire graph of data will be selected and retrieved to the querying site.

The alternative (which we will call S2) is to process the query iteratively, only retrieving edges if and when they are to be traversed. In this case, we only retrieve the data that is actually needed during the query processing, but at the expense of many more subqueries, and most likely slower query processing, assuming that network latency is the main bottleneck. Unfortunately, even if S2 only retrieves the data that is needed, for some worst case queries (e.g. a query where the regular expression contains patterns such as $.^*$), the full graph of data could still be retrieved.

This clearly indicates that in terms of network traffic cost, a worst-case complexity analysis will be unconclusive: for both of the considered techniques, some queries will result in the entire graph of data being retrieved.

However, we expect that practical, real-world queries would not result in this worst-case scenario, and the main cost factors between these two techniques are different: in one case, we only send out one query to the data sources, but potentially retrieve much more data than we need, whereas in the other case we

cause large numbers of subqueries, but retrieve only the data that we need. In addition, the cost of a query depends on how many data sources we send it to, and the cost of retrieving data depends on the replication of this data across different data sources.

In the following section, we analyze this trade-off empirically, using real-world data and queries from the biomedical domain.

4 Cost Comparison on Real-world Queries

4.1 Dataset and queries

In order to conduct an empirical study, we acquired a real-world dataset from the biomedical domain, with some meaningful queries to apply to this data. The dataset is a graph of knowledge automatically extracted from a corpus of pubmed abstracts [21]. The graph has approximately 50,000 nodes and 340,000 edges. The nodes identify concepts such as molecules, genes, or animal species, and the edges represent relationships, such as a molecule activating a gene.

The queries express meaningful associations between biological entities. They are multi-source RPQ (i.e. regular expressions with no given starting node), and are listed in Table 1, along with the number of solution pairs for each query. The queries are the same ones used in [15]; the queries and the dataset were kindly provided by the authors of that study.

As the graph has 50,000 nodes, we can create 50,000 single-source queries from each regular expression (multi-source query). However, for many nodes the relationship expressed by the query simply does not make sense (e.g. *phosphorylation* happens to proteins or other molecules, but not to rabbits or humans), which in practice means that for most nodes there will be no adjacent edges matching the beginning of a query path, and therefore the cost of evaluating the query will be basically nil. For the queries of interest here, less than 2% of the nodes were valid starting points. The exact number of valid starting points for each query is given in the last column of table 1.

When evaluating the cost of queries, we will restrict our analysis to these valid starting points. Obviously, calculating the average cost when it is nil 99% of the time would produce results of little value.

In order to compare the cost of strategies S1 and S2 for these queries, one way would be to evaluate them in a distributed setting, and vary the parameters of this distributed setting. However, for these strategies, the distribution parameters affect the cost of each subquery, which in turn determine the cost of the query execution. Specifically, the number of sources and the network topology (the latter only relevant in a peer-to-peer scenario) will determine the cost of distributing the subqueries to the data sources (i.e. *broadcasting* the subqueries), whereas the data *replication* across these sources will increase the volume of data returned to the query originator (i.e. the amount of *unicast* data).

Expressing the costs of broadcasting or unicasting data as a function of the network parameters will then allow us to obtain the actual cost of each query as a function of these parameters.

	Query	Multi-source solution pairs	Start nodes with non-zero cost
q1	C^+ "acetylation" A^+	1710	477
q2	C^+ "acetylation" I^+	20	477
q3	C^+ "methylation" A^+	0	477
q4	C^+ "methylation" I^+	0	477
q5	C^+ "fusions" P	0	477
q6	"fusions" A^+	8	2
q7	A^+ "receptor" P	0	731
q8	I^+ "receptor" P	0	366
q9	A A^+	80905	711
q10	I I^+	2118	354
q11	C E	249	364
q12	A^+ I^+	49638	711

Edge labels ('|' means disjunction):

C =	{interaction | interactions | binding | complex | interacting | complexes | interacts}
A =	{activation | activity | production | induction | overexpression | up-regulation | induces | activates | increases}
I =	{down-regulation | inhibits | inhibited | inhibitor | inhibition}
E =	{expression | overexpression | regulates | up-regulation | expressing}
P =	{dephosphorylates | dephosphorylated | dephosphorylate | dephosphorylation | phosphorylates | phosphorylated | phosphorylate | phosphorylation}

Table 1: Biomedical queries: regular expression, number of solutions (multi-source), number of valid start nodes.

4.2 Broadcast and Unicast Costs

For this purpose, we first compute the number of unicasts and broadcasts for each of our example queries. In the following analysis, we consider each symbol (edge label or node identifier) transmitted in a query or response as the unit of cost for message traffic. The true cost would be obtained by multiplying our values by the number of bytes per symbol, and adding some overhead for the message headers.

Cost of S1 The processing of a query by S1 requires a single initial query (broadcast) to retrieve all the data (a subset of G_D) potentially needed to process the RPQ.

This data is the set of edges whose labels are found in the query, and the length of the initial broadcast query is therefore the number of distinct labels in the query.

The data matching the query is then returned by "point-to-point" messages (unicasts). The amount of data to be transferred is the number of matching

edges to the initial broadcast, multiplied by the length of each edge: we consider that an edge is expressed as 3 symbols, two node identifiers and an edge label.

It is important to notice that for strategy S1, the cost does not depend on the query start node, and is even the same for a single-source or a multi-source query.

Cost of S2 In strategy S2, the PAA is executed locally, accessing the remote data through iterative broadcast subqueries.

During the search of the product automaton, at each node there is a broadcast search to find neighbours of the current graph node. Each time, the broadcast query indicates the current node and the labels of the potential outgoing edges, which are the symbols associated with the outgoing transitions from this automaton state.

The amount of data to be broadcast is the sum of the lengths of all these individual queries. The data returned as unicast messages is the set of edges in G_D that match them.

We assume a simple optimization whereby two identical broadcast queries that are made at different points of the algorithm will result in a single one being processed over the network, and the second time its results are obtained from a local cache.

Unlike strategy S1, for strategy S2 each single-source query (using the same regular expression and different start nodes) will have a different cost.

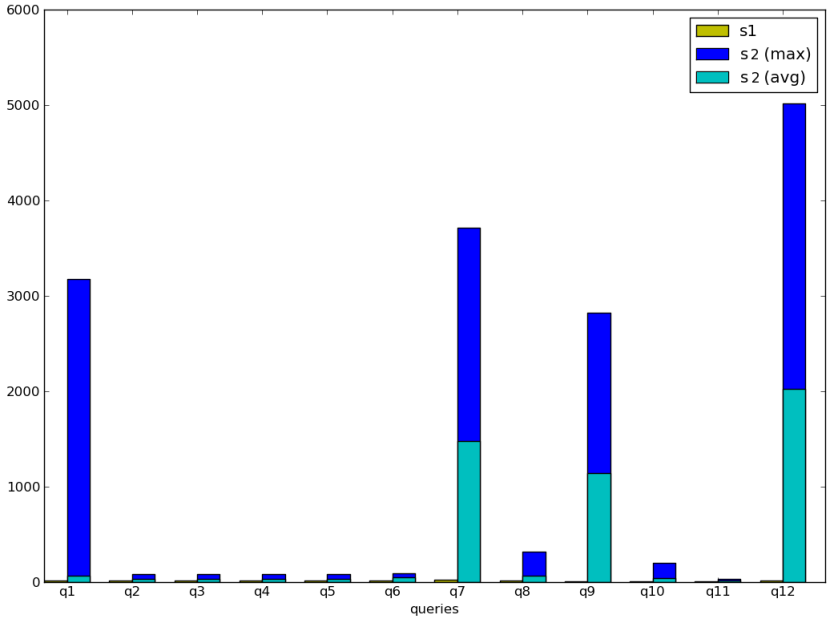
4.3 Results

Figures 1a and 1b summarize the values obtained for the above cost functions, with our biomedical queries.

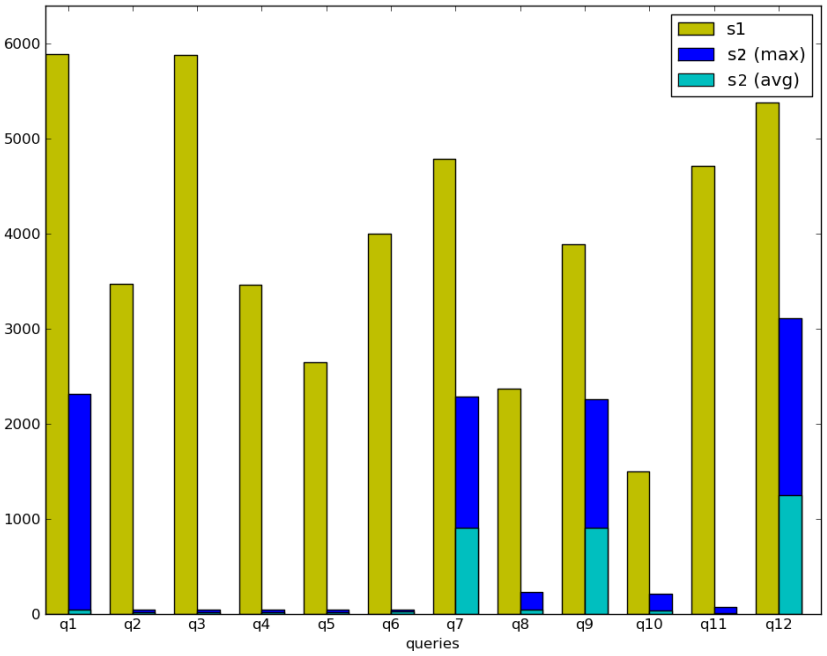
For each regular expression, we compare side-by-side the cost of S1 (which is the same for all the single-source queries) and the costs of S2: we show here the mean and the maximum cost. Note that the mean is calculated only for valid starting points, i.e. it is the mean of all non-zero costs.

These figures illustrate well the trade-off between broadcasts and unicasts for the two strategies. Strategy S1 always requires a minimal amount of data to be broadcast, but also consistently retrieves fairly large amounts of data via unicast. Strategy S2 has typically higher broadcast costs and much lower unicast costs, and is also much more variable. We note that the queries considered here are all quite selective, in the sense that they only retrieve a small fraction of the data graph. S1 retrieves between 0.2% and 0.8% of the graph, whereas S2 retrieves less than 0.1% of the graph in almost every case.

However, due to this trade-off and the high variations in the cost of S2, it is unclear which strategy is generally preferable. In the following, we examine this trade-off in analytical terms, in relation to the parameters that determine the costs of broadcast and unicast messages.



Broadcasts



Unicasts

Fig. 2: Amount of data to be transferred by broadcast /unicast for the evaluation of the biological queries.

4.4 Cost functions

In order to express and compare the costs of S1 and S2, we introduce the following notations:

- the number of distinct labels in a query q is denoted by $Q_{lbl}(q)$,
- the number of matching edges, or rather the amount of data representing this information, is denoted by $D_{S1}(q, G_D)$,
- $Q_{bc}(q, G_D)$ denotes the total amount of data that is *broadcast* with strategy S2,
- $D_{s2}(q, G_D)$ denotes the total amount of data *unicast* for S2.
- The data is replicated on average K times, where $K = k.N_p$: we denote by N_p the network size and k is then the fraction of peers that replicate each data resource.

In the above functions we have indicated the dependencies of these quantities on q and G_D as function arguments. In the following we leave these function arguments out in order to improve readability.

In a Peer-to-peer context, the cost (number of messages) of a broadcast in a connected network with N_c edges is between N_c (best case) and $2.N_c$ (worst case). If the average (outgoing) node degree in the network graph is d, then N_c can be approximated as $d.N_p$. Ignoring the protocol overhead for each message, we can therefore approximate the cost of broadcasting b bytes of data as $2.d.N_p.b$.

In a Linked Data context, a client needs to connect directly to each data sources: if there are N_p data sources, then this amounts to the above cost, with $d = 0.5$.

We obtain the following cost functions:

$$\begin{aligned} cost_{S1}(q, G_D) &= 2.N_c.Q_{lbl} + k.N_p.D_{s1} \\ &= 2.d.N_p.Q_{lbl} + k.N_p.D_{s1} \\ &= N_p(2.d.Q_{lbl} + k.D_{s1}) \end{aligned} \tag{1}$$

$$\begin{aligned} cost_{S2}(q, G_D) &= 2.N_c.Q_{bc} + k.N_p.D_{s2} \\ &= N_p(2.d.Q_{bc} + k.D_{s2}) \end{aligned} \tag{2}$$

4.5 Query Execution Strategy Choice

Using equations 1 and 2, we can establish the following condition, determining whether S2 or S1 is preferable :

$$\begin{aligned} & 2.d.Q_{lbl} + k.D_{s1} < 2.d.Q_{bc} + kD_{s4} \\ \Leftrightarrow\ & k.(D_{s1} - D_{s2}) < 2.d(Q_{bc} - Q_{lbl}) \\ \Leftrightarrow\ & \frac{k}{d} < 2\frac{Q_{bc} - Q_{lbl}}{D_{s1} - D_{s2}} \end{aligned} \tag{3}$$

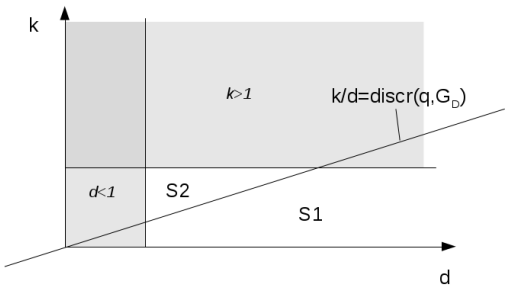


Fig. 3: Optimality of S1 and S2 depending on k, d and the query-dependent discriminating function.

In the following, we will use the notation:

$$discr(q, G_D) = 2\frac{Q_{bc} - Q_{lbl}}{D_{s1} - D_{s2}}$$

Equation 3 provides a discriminating condition to choose between S1 and S2, independent of the network size. Parameters d and k characterize the network topology and the data distribution within this network. Higher values of d (denser networks) increase the cost of broadcasts, therefore favouring strategy S1, that broadcasts less data, whereas higher values of k (higher data replication rates) increase the cost of retrieving data (each data resource retrieved comes in more copies), therefore favouring S2, which only retrieves the data necessary to execute the PAA.

We also know that $k < 1 < d$, because $k > 1$ would mean that every peer has multiple copies of the full graph of data, and if $d < 1$ the network graph cannot be connected[3].

This gives us the following discriminating conditions (Fig. 3):

- If $Q_{bc}(q, G_D) \leq Q_{lbl}(q, G_D)$ then S2 is necessarily optimal. The trivial case is where the query starting point is not valid, and this may also happen with very long and complex queries.
- If $Q_{bc}(q, G_D) > Q_{lbl}(q, G_D)$, then in the 2-dimensional space of values for k and d, S2 is optimal in a triangle bounded by the lines of equations $k = 1$, $d = 1$ and $\frac{k}{d} = discr(q, G_D)$.
- For any other values of k and d that fulfil the condition $k < 1 < d$, S1 is optimal.
- Note that if $discr(q, G_D) > 1$, then S1 is necessarily optimal, because the triangle described above does not intersect with the region where $k < 1 < d$.

[3] with the exception of the Linked Data setting, which is equivalent to $d = 0.5$

For our example biomedical queries, of the 5622 single-source queries with non-zero cost[4]), in 42 cases S2 is necessarily optimal, and for the 5580 others, either S1 or S2 will be optimal depending on the network parameters.

The problem is that while we can immediately identify queries with zero cost, and have immediate access to $Q_{lbl}(q)$ (the number of distinct labels in the query), the other parameters are not readily accessible. Therefore, $discr(q, G_D)$ cannot be calculated without actually executing the query over the data of interest, which means that despite this analysis we still cannot tell, *a priori*, which strategy to choose.

In the following section, we address this problem with methods to estimate the different components of $discr()$, from simple network queries and a sample of G_D.

5 Query Cost Estimation

As we have found above, the parameters that determine the optimality of S1 and S2 are the following: N_p, N_c, k, and the elusive discriminating function $discr(q, G_D)$, which measures the selectivity of q over G_D. The first three parameters can be obtained or estimated from network queries:

- N_p, network size: in a P2P setting, this parameter can be obtained by broadcasting a "ping" query, where each recipient simply responds with an acknowledgement. In a Web setting, the number of data sources is known.
- N_c, number of network connections (N_c is only relevant to the P2P setting): can be obtained by broadcasting a query requesting each peer's number of active network connections. The sum of responses will be $2 * N_c$.
- k, data replication rate: can be estimated by querying for a small number of known data resources, and counting the average number of responses. This will yield an estimate of K (data replication factor), which can then be divided by N_p to obtain k. The accuracy of this estimate depends on using a representative sample of resources. Querying for more resources incurs more costs but will improve the estimate.

We now turn to the components of the "query selectivity" function *discr*:

- $Q_{lbl}(q)$ can be trivially obtained from the query: it is the number of distinct edge labels appearing in the regular expression.
- $D_{s1}(q, G_D)$ can be estimated by counting the label frequencies on a sample of the data, and multiplying by the total number of edges $|E|$. $|E|$ can be estimated by a broadcast query requesting a count of the distinct resources stored at each site, then dividing by the expected replication K.
- $Q_{bc}(q, G_D)$ and $D_{s2}(q, G_D)$ depend directly on the *selectivity* of the query q (which is known) over G_D (which is only partially known). Ideally we would like to estimate these parameters from a *sample* of G_D. This is the focus of this section.

[4] Altogether we could apply each of the 12 multi-source queries to 50,000 nodes, yielding 600,000 single-source queries.

Simply evaluating q over a sample of the G_D is unlikely to give us any indication of the true cost of the query over the full graph, if only because it is highly unlikely that the start node will be present in the sample.

Our approach is based on statistical models of the graph of data. The idea is to compute a model of the graph from a sample, then re-generate a larger graph from the model: evaluating q on this synthetic graph can then give us an estimate of the cost of evaluating q on G_D.

5.1 Statistical Graph Models

Several data structures have been proposed to *summarize* graph data [10, 8, 5], but these techniques mainly give indications of whether paths exist, with no direct application to cost estimation. In order to estimate the number of different paths, we use *statistical models* of the graphs, which give us the *probability* of encountering an edge with a given label.

Binomial Random Graph The first model that we investigate is based on the binomial random graph model (or Gilbert model [9]), where for any pair of nodes (v_1, v_2), there is a probability p that the edge (v_1, v_2) exists. Extending this model to labelled graphs, for any label a, each edge (v_1, a, v_2) exists with a probability $p(a)$.

The probabilities $p(a_i)$ for the different edge labels can be estimated by frequency counts. Using this model, the cost of a query can be estimated by executing the PAA replacing the access to the data graph with a function that randomly generates edges using the binomial distribution.

Bayesian-Binomial Random Graph The disadvantage of the above model is that it completely ignores the graph structure, in the sense that adjacent edges have independent probabilities of existing. In fact, in a real-world graph, it is likely that the presence and labels of adjacent nodes are correlated, due to the semantics of the relationships that they represent.

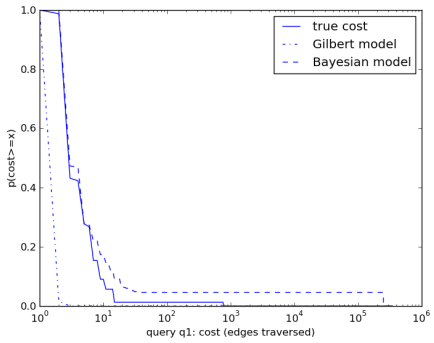
A more elaborate model should therefore estimate the probabilities of edges conditional to the existence (and labels) of adjacent edges. Although such a "static" model is difficult to describe, we can use a "generative" model: as above we apply the PAA and replace the access to the data graph with a function that randomly generates edges, except this time we generate edges using probabilities conditional to the label of the edge that brought us to this node.

5.2 Cost Estimation

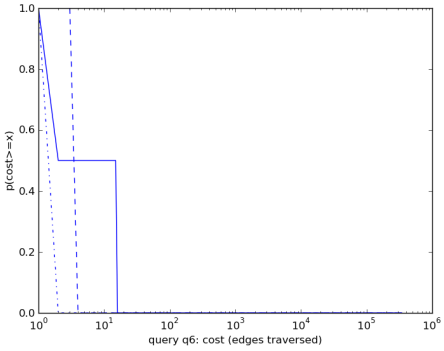
Using the above models, we estimated the cost of our biological queries, and compared these estimates with their true cost.

As we have noted previously, for single-source queries using S2, the cost factors (the amount of data that is broadcast, and the amount of data retrieved by unicast) vary wildly depending on the start node. However, they are also

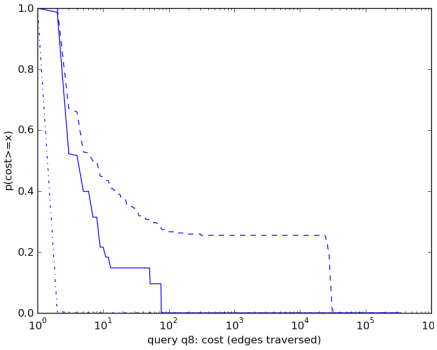
highly correlated, and we will focus here on results for the number of edges traversed during query execution, which can be considered an (inverse) measure of the query selectivity. Since applying the same path query to different (valid) starting points yields very different costs, we attempt to estimate the probability *distribution* of these costs, rather than the exact cost for a given starting point. For each query, we compared the real distribution (frequencies) of costs for the different start nodes, with the distribution of costs obtained for many runs of our estimation algorithms. Specifically, since the graph has 50,000 nodes, we have 50,000 true costs, and we also ran 50,000 runs of the algorithms for the models. Recall that 99% of the time, the cost is nil; this was true for the models as well, within one or two percent.



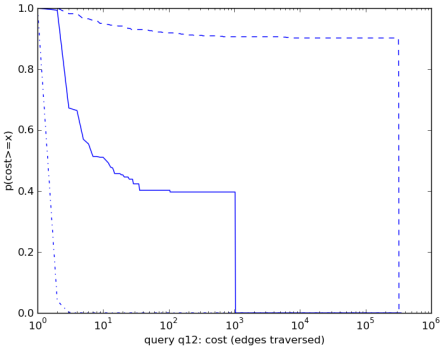
Tail Distribution for query q1.



Tail Distribution for query q6.



Tail Distribution for query q8.



Tail Distribution for query q12.

Fig. 5: Tail Distributions for different queries: true distribution, and estimates based on the two types of statistical graph models. Only 4 queries are shown, the others are comparable. Note the logarithmic scale on the x-axis.

We constructed the models using the full graph of data: this allows us to evaluate the technique itself, without the noise caused by imperfect sampling[5].

In order to compare the probability distributions of the true costs and the models, we have plotted the three tail distributions (complementary CDF) for each query (Fig. 5).

As there are no established metrics or baselines to evaluate these models, so we will limit ourselves to an informal explanation of these resulting figures. Across all queries, we observe that the first model consistently underestimates costs, whereas the Bayesian model tends to overestimate them, although there are exceptions (e.g. query 6). The reason why the Gilbert model underestimates the likelihood of paths is that it ignores the structure of the graph. As we have mentioned before, the labels of the edges incident to a given node are all consistent with the node semantics, and thus correlated. As the model considers the labels of different edges to be independent, it underestimates the probability of repeating edge labels. Paths estimated by the model are thus much shorter than in the real data.

This is precisely the point of using conditional probabilities in the Bayesian model. The Bayesian model estimates the probabilities of outgoing edges of a node conditionally to the label by which we reached this node.

For example, we could be following paths labelled a^*bb, where the label b might be very rare, but such labels may be clustered together due to the semantics of the relationship b. This would mean that the probability of an edge b existing between two arbitrary nodes v_1 and v_2 may be very low, but might be much higher if we know that v_1 has an incoming edge labelled b.

Where the Bayesian model falls short of perfection is that although it may produce good estimates of the number of outgoing edges from a given node, it then picks the targets of these edges at random, ignoring other structural properties of the graph such as *clustering* (in an undirected graph) or edge *transitivity*, the equivalent in a directed graph. These properties mean that in real-world graphs, if two nodes v_1 and v_2 have a common neighbour v_3, they are more likely to be themselves connected than would be expected by random chance. This implies that paths with a common origin will tend to merge together and explore fewer nodes (than would be expected in a random graph without those structural properties).

6 Conclusion

We studied the problem of evaluating regular path queries on distributed data, where data is not localized, and is accessed through broadcast or unicast messages.

[5] Ideally, a *representative* sample of the data should produce the same graph label frequencies, and therefore the same models. Variations due to imperfect sampling are not our main concern here. We defer to future work the evaluation of approximate models obtained using only a sample of the data.

For this setting, we have compared analytically and experimentally two general approaches, the centralized "top-down" and "bottom-up" approaches (S1 and S2, respectively). We found that S1 generates most of its cost by retrieving more data than necessary, while S2 only retrieves the data that it needs, but requires more broadcasts to locate this data during its execution.

Therefore, S2 performs better on more selective queries, where S1 is very wasteful. However, until now there were no known techniques to estimate the selectivity of a query without executing it on the dataset of interest. In order to address this problem, we proposed a query cost estimation approach based on two classes of statistical graph models. This ultimately provides a way to choose between S1 and S2, and to evaluate the cost of processing a query before potentially flooding the network with an excessive amount of traffic.

References

1. S. Abiteboul and V. Vianu. Queries and computation on the web. In F. Afrati and P. Kolaitis, editors, *Database Theory — ICDT '97*, volume 1186 of *Lecture Notes in Computer Science*, pages 262–275. Springer Berlin Heidelberg, 1997.
2. S. Abiteboul and V. Vianu. Regular path queries with constraints. In *Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 122–133. ACM, 1997.
3. P. Barceló Baeza. Querying graph databases. In R. Hull and W. Fan, editors, *PODS*, pages 175–188. ACM, 2013.
4. D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Y. Vardi. Reasoning on regular path queries. *SIGMOD Record*, 32(4):83–92, 2003.
5. Q. Chen, A. Lim, and K. W. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 134–144, New York, NY, USA, 2003. ACM.
6. M. P. Consens and A. O. Mendelzon. The G+/Graphlog visual query system. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, pages 388–, New York, NY, USA, 1990. ACM.
7. I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A graphical query language supporting recursion. In *ACM SIGMOD Record*, volume 16, pages 323–330. ACM, 1987.
8. M. F. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In S. D. Urban and E. Bertino, editors, *ICDE*, pages 14–23. IEEE Computer Society, 1998.
9. E. N. Gilbert. Random graphs. *The Annals of Mathematical Statistics*, 30(4):1141–1144, December 1959.
10. R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. Technical Report 1997-50, Stanford InfoLab, 1997.
11. A. Halevy, Z. Ives, J. Madhavan, P. Mork, D. Suciu, and I. Tatarinov. The Piazza peer data management system. *Knowledge and Data Engineering, IEEE Transactions on*, 16(7):787–798, July 2004.
12. A. Harth, K. Hose, M. Karnstedt, A. Polleres, K.-U. Sattler, and J. Umbrich. Data summaries for on-demand queries over linked data. In *Proceedings of the*

19th International Conference on World Wide Web, WWW '10, pages 411–420, New York, NY, USA, 2010. ACM.
13. O. Hartig, C. Bizer, and J.-C. Freytag. Executing sparql queries over the web of linked data. In A. Bernstein, D. Karger, T. Heath, L. Feigenbaum, D. Maynard, E. Motta, and K. Thirunarayan, editors, *The Semantic Web - ISWC 2009*, volume 5823 of *Lecture Notes in Computer Science*, pages 293–309. Springer Berlin Heidelberg, 2009.
14. A. Koschmieder. Cost-based optimization of regular path queries on large graphs. In W.-T. Balke and C. Lofi, editors, *Grundlagen von Datenbanken*, volume 581 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2010.
15. A. Koschmieder and U. Leser. Regular path queries on large graphs. In A. Ailamaki and S. Bowers, editors, *SSDBM*, volume 7338 of *Lecture Notes in Computer Science*, pages 177–194. Springer, 2012.
16. G. Ladwig and T. Tran. Linked data query processing strategies. In *Proceedings of the 9th International Semantic Web Conference on The Semantic Web - Volume Part I*, ISWC'10, pages 453–469, Berlin, Heidelberg, 2010. Springer-Verlag.
17. A. O. Mendelzon, G. A. Mihaila, and T. Milo. Querying the world wide web. *Int. J. on Digital Libraries*, 1(1):54–67, 1997.
18. A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. In P. M. G. Apers and G. Wiederhold, editors, *VLDB*, pages 185–193. Morgan Kaufmann, 1989.
19. G. Navarro. Pattern matching. *Journal of Applied Statistics*, 31(8):925–949, 2004. Special issue on Pattern Discovery.
20. W. Nejdl, B. Wolf, C. Qu, S. Decker, M. Sintek, A. Naeve, M. Nilsson, M. Palmér, and T. Risch. EDUTELLA: a P2P networking infrastructure based on RDF. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 604–615, New York, NY, USA, 2002. ACM.
21. C. Plake, T. Schiemann, M. Pankalla, J. Hakenberg, and U. Leser. Alibaba: Pubmed as a graph. *Bioinformatics*, 22(19):2444–2445, 2006.
22. M. Saleem, A.-C. N. Ngomo, J. X. Parreira, H. F. Deus, and M. Hauswirth. DAW: Duplicate-aware federated query processing over the web of data. In *International Semantic Web Conference*, pages 574–590. Springer, 2013.
23. M. Shoaran and A. Thomo. Fault-tolerant computation of distributed regular path queries. *Theor. Comput. Sci.*, 410(1):62–77, 2009.
24. D. Suciu. Query decomposition and view maintenance for query languages for unstructured data. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *VLDB*, pages 227–238. Morgan Kaufmann, 1996.
25. J. Umbrich, A. Hogan, A. Polleres, and S. Decker. Link traversal querying for a diverse web of data. *Semantic Web Journal*, 2014.
26. M. Y. Vardi. The complexity of relational query languages (extended abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, pages 137–146, New York, NY, USA, 1982. ACM.
27. N. Yakovets, P. Godfrey, and J. Gryz. Towards query optimization for SPARQL property paths. *CoRR*, abs/1504.08262, 2015.